periodica polytechnica

Electrical Engineering
52/3-4 (2008) 163–176
doi: 10.3311/pp.ee.2008-3-4.05
web: http://www.pp.bme.hu/ee

RESEARCH ARTICLE

Static Execute After algorithms as alternatives for impact analysis

Judit Jász

Received 2008-07-22

Abstract

Impact analysis plays an important role in many software engineering tasks such as software maintenance, regression testing and debugging. In this paper, we present a static method to compute the impact sets of particular program points. For large programs, this method is more effective than the slightly more precise slicing. Our technique can also be used on larger programs with over thousands of lines of code where no slicers can be applied since the determination of the program dependence graphs, which are the bases of slicing, is an especially expensive task. As a result, our method could be efficiently used in the field of impact analysis.

Keywords

impact analysis · execute after and execute before relations · program slicing

Acknowledgement

The author wishes to thank the co-porative work of Árpád Beszédes, Tamás Gergely, Tibor Gyimóthy and Vaclav Rajlich. This work was in part supported by Hungarian National grants no. RET-07/2005 and OTKA K-73688.

Judit Jász

Department of Software Engineering, University of Szeged, Árpád tér 2, H-6720 Szeged, Hungary

e-mail: jasy@inf.u-szeged.hu

1 Introduction

During software engineering, as a program is evolving it becomes more and more complex and hard to predict what other changes are induced by a simple change. In addition, as the size of the program is growing it is more expensive to repeat the regression tests responsible for quality after a change. The task of impact analysis is to discover what other program points are affected by a change of a particular program point, or it can even be to safely reduce the necessary testing steps by taking into account the changes and thus making the continually running regression tests more effective.

A rational solution of impact analysis is determining the set of instructions impacted by the change with a forward slice starting from the changed program point[1] [12]. However, the methods that omit the expensive costs of slicing work with smaller and more easily identifiable dependency graphs, so they are more often used in practice [2, 10, 19]. This is usually possible because during impact analysis it is not expected to discover statement level connections, it is more common to give the results at the procedure level.

If we only have the call graph, which contains only the relative easily computable call dependencies, we might have an unsafe approximation for the set of procedures which may be affected by the change [10]. In addition, there are solutions which combine these two approaches; they work on a smaller dependence graph and make the results determined on the call graph more precise with program slicing [20].

Another relevant issue of impact analysis – as of any fields of software analysis – is whether the results should be determined by using dynamic or only static information. The main disadvantage of the static approaches is said to be their conservatism because the size of the dependence graphs can grow dramatically in many cases[2]. The disadvantages of the dynamic approaches reside in the process and storage of the execution traces and in the fact that the results depend on how the traces

[1] A forward slice of a program point consists of the program points whose execution is influenced by the given program point.

[2] For example, in the call graph the ambiguous targets of function pointers and virtual function calls can increase the size.

can reflect the general behaviour of the given program. On the basis of the facts described above, the investigation of large program systems with dynamic analysis is a very expensive task. Papers [2, 19, 20] deal with dynamic impact analysis, while papers [4, 10] deal with static impact analysis.

In this paper, our aim is to give an alternative way to determine the static impact sets of procedures by applying *Static Execute After* relation among them. The approach is motivated by Apiwattanapong [2], who introduced the notion of *Execute After* relation and applied it in dynamic impact analysis. We give a graph representation where a suitable traversal gives this relation. In addition, we give two algorithms that compute the sets of these relations. The impact sets of procedures can be approximated by these sets. We prove with experimental results that the computed sets can approximate the sets of sliced procedures computed by a precise slicer, and in this way, we can approximate the impact sets as well. Our method approximates the results of the slicer with high precision, over 80% in most cases. On average, the precision declines only very slightly, by some 4%. Since the introduced algorithms are based on much less dependencies than the slicer and they work on a more compact graph representation, they are more convenient to use especially in the case of large programs. The use of a slicer is strongly affected by the size of the program. While due to the size of graph representation the slicing of thousands of lines of code is unrealizable, our representation is easy to build and use. Although the introduced technique is language independent, since it only needs the call graphs and the control flow graphs of the analyzed programs, we made our experiments on C/C++ programs so that the results could be comparable with the results of the CodeSurfer [13], a commercial program slicing tool marketed by GrammaTech Inc.

The paper is organized as follows. Section 2 reviews the related works. The Static Execute After relation is introduced in Section 3. In this section, we give an appropriate graph representation for the computation of this relationship and we give two algorithms for its computation. In Section 4, we confirm with experimental results that the impact sets constructed by a static slicer can be efficiently approximated by the impact sets determined by the Static Execute After relationship. In addition, the graph representation needed for the Static Execute After relationship is more compact than the system dependence graph essential for slicing and in this way its use is more efficient in the case of large programs. The expected property of our method is to find all the dependencies found by the slicer and so, the recall value will be 100%. The fact that this property is not realized in every forward slice is an interesting by-product of our experimental results. We investigate the reason of this in Section 5. Section 6 discusses the conclusion.

2 Related works

One of the first methods for calculating impact sets is built only on the call graph of the program [10]. The impact set of a modified procedure of the examined program consists of the procedures which are reachable from that modified procedure in the call graph. This kind of technique is not only imprecise but it is also incomplete in the sense that it does not suppose that the modified procedure can influence the caller procedures. Although the inaccuracy of this method is evident and criticized by other researchers [21], there are no practical measurements to confirm the rate of its inaccuracy.

The algorithm `PathImpact` by Law and Rothermel [18, 19] deals with dependence based dynamic impact analysis where compressed running information is generated by the executions of the instrumented version of the program. By applying forward and backward traversal on the result, we can create the before and the after execution sets of a given procedure. These sets are considered as the impact sets of the procedure. Orso *et al.* introduced `CoverageImpact` analysis [20] where the program executions affected by the modified procedures are selected first, then the set of procedures covered by these selected executions are collected and finally, we determine which procedures are affected by the static forward slicing starting from the modified procedures. The impact set of the given procedure consists of the intersection of the set of covered procedures and the set of procedures inserted into slicing.

Apiwattanapong *et al.* compared these two methods with a third one given by them [2]. In the practical comparison they examined different versions of the programs and they created the dynamic data with the help of the test suites of these programs. They classified the methods according to their performance and precision only. This resulted in the fact that the newly introduced method, which is based on the determination of the execute after relation of the procedures, is at least as precise as the `PathImpact` method and at least as efficient as the `ChangeImpact` analysis.

Beszédes *et al.* defined the `DynamicFunctionMetric` [7] between procedures with the previously introduced execute after relation thus giving a more precise way to construct the procedure level impact sets. Their empirical results show that if we consider the sets of procedures that are at different DFC distances from a given procedure, then the set which is closer to the given procedure more probably contains really dependent procedures than a farther one. For calculating precision and completeness values, they used the results of the Jadys dynamic slicer [23] extended to procedure level.

Badri *et al.* [4] deal with giving static impact sets built on dependencies. The starting point of their approach is similar to the one presented by us. To determine the static impact sets they introduce the *control call graph* program representation, which is practically created from the control flow graph and the call graph by leaving out the nodes that do not influence the execution of the procedure calls[3]. Badri's algorithm determines the

[3]This representation is very similar to the CCFG representation introduced by us apart from the difference that we do not keep nodes which represent con-

compact series of procedure calls for each procedure by introducing different notations for the calls in the iteration and in the different branches of the conditions. In the practical analysis of the algorithm, they try to compare the introduced technique only with the impact sets reachable by the call graph, but it is not clear how the precision of the results are calculated.

In this paper, we show how imprecise the impact sets given only by the call graph or by other easily computable graphs are, but the main part of our work deals with the use of the static execute after relation. Although similarly to Apiwattanapong *et al.* [2], we use the concept of precision to compare the results, this value does not give the proportion of the impact set and the real size of the program. Instead, we compare the impact sets given by us with the impact sets given by static slicing as Beszédes *et al.* [7] did while examining the relationship between the DFC metrics and the dynamic slice. We give a static approach for impact sets, but we confirm our statements with much more precise results not only by examining selected procedures like Badri *et al.* [4] did, but by summarizing the results of all procedures of the examined programs.

This paper is based on our earlier papers [8, 16], but besides, it introduces the Static Execute After relations from the point of view of impact analysis. The main contributions of this paper are the followings: it

- improves the algorithm introduced by [8],
- gives the formal description of Static Execute After algorithm, which is the pair of the algorithm in [16],
- compares the previously mentioned algorithm types,
- gives the detailed introduction of the Interprocedural Component Control Flow Graph,
- supplements the results of [16] with further comparisons,
- demonstrates that the handling of arbitrary control flows in the slicer requires the modification of both the program graph representation and the slicing algorithm. Our experiments revealed that the slicing algorithm was imprecise in the forward slicer used in our measurements. We introduced this bug and its causes in Section 5. In this paper, we examine the effect of this conservative slicing method on the results.

3 Computation of SEA

Our work is motivated by the *Execute After* relation introduced by Apiwattanapong *et al.* [2]. They used this relation to determine the dynamic impact sets of procedures. According to the definition, the procedures f and g are in Execute After relation if and only if any part of g is executed after any part of f in any of the selected set of executions of the program.

As a static counterpart of this approach we define the *Static Execute After* (SEA) relation. We can say that $(f, g) \in$ SEA if and only if it is possible that any part of g may be executed after any part of f[4].

As the notion of the backward slice is the dual of the forward slice, the *Static Execute Before* (SEB) relation can be determined as a dual counterpart of the SEA. The procedures f and g are in SEB relation if and only if it is possible that any part of g may be executed before any part of f.

According to Apiwattanapong *et al.* [2] and Beszédes *et al.* [7] the formal definition of the SEA relation is the following:

$$\text{SEA} = \text{CALL} \cup \text{SEQ} \cup \text{RET}[\cup \text{ID}],$$

where

$$\begin{aligned}
(f, g) \in \text{CALL} &\iff f \text{ (transitively) calls } g, \\
(f, g) \in \text{SEQ} &\iff \exists h : f \text{ (transitively) returns into } h, \text{ and after that } h \text{ (transitively) calls } g \\
(f, g) \in \text{RET} &\iff f \text{ (transitively) returns into } g
\end{aligned}$$

or rather the ID is the identic relation that can optionally be the part of SEA, since a slice also contains the criterion itself and every change in a function f can affect any part of f from the impact analysis point of view.

It is easy to see that the union of the given relations are suitable to determine the SEA relation, if we rephrase the above definition. So $(f, g) \in$ SEA if and only if there is a path on the CFG where any of the f_{entry} events comes before any of the g_{exit} events. If we consider each procedure event in pairs, where each exit entry is related to a given entry, we get three cases. The path can contain the event pairs of procedure f and g in the following sequences[5]:

- f_{entry}, g_{entry}, g_{exit}, f_{exit}, in this case procedure f (transitively) calls procedure g.
- f_{entry}, f_{exit}, g_{entry}, g_{exit}, in this case there is a procedure h, which first (transitively) calls procedure f and after (transitively) calls procedure g.
- g_{entry}, f_{entry}, f_{exit}, g_{exit}, in this case procedure f (transitively) returns into procedure g.

Since f_{entry} always comes before f_{exit}, the identic relation can also be the part of the SEA relation.

3.1 The ICCFG graph

We have to build a suitable program representation to determine the SEA relations. The traditional call graph representation [22] is not sufficient, since it says nothing about the order

ditions and branches and the given nodes are contracted by their connected components.

[4] Entering into a procedure, or leaving a procedure are also the events of the procedure.

[5] f_{entry}, g_{entry}, f_{exit}, g_{exit} and g_{entry}, f_{entry}, g_{exit}, f_{exit} are impossible scenarios.

```
#include <iostream>
#include <math.h>
#include <string>
using namespace std;

typedef struct Func{
  string* name;
  double (*func)(double);
  struct Func* nextFunc;
} *Function;

void init (Function* list){
  Function tmp = (Function) malloc (sizeof(struc Func));
  tmp->name = new string("sin");
  tmp->func = sin;
  tmp->nextFunc = NULL;
  *list = tmp;

  tmp=(Function) malloc (sizeof(struct Func));
  tmp->name = new string("cos");
  tmp->func = cos;
  tmp->nextFunc = *list;
  *list = tmp;
}

void deleteName(Function elem){
  if (elem->name)
    delete(elem->name);
}

void deleteFunctions (Function list){
  Function tmp=list;
  while (tmp){
    Function next = tmp->nextFunc;
    deleteName(tmp);
    free(tmp);
    tmp = next;
  }
}

void readInputs(string* inputFunction, double* input){
   cout << "What is the selected function?  ";
   cin >> *inputFunction;
   cout << "What is the argument?  ";
   string arg;
   cin >> arg;
   *input = atof(arg.c_str());
}

bool checkInputs(Function list, string input, Function* selected){
  Function tmp = list;
  while (tmp){
    if ((*tmp->name).compare(input) == 0){
      *selected = tmp;
      break;
    }
    tmp = tmp->nextFunc;
  }
  return *selected != NULL;
}

int main(int argc, char *argv[]){
  Function functionList = NULL;
  init(&functionList);

  string inputFunc;
  Function selectedFunc = NULL;
  double input = 0.0;
  bool isAnyInput = false;
  while (!isAnyInput){
    readInputs(&inputFunc, &input);
    isAnyInput = checkInputs(functionList, inputFunc, &selectedFunc);
  }

  cout <<inputFunc <<"("<<input<<")=" <<selectedFunc->func(input)
                                      << endl;
  deleteFunctions(functionList);
}
```

Fig. 1. Example program

of the procedure calls within a procedure. On the other hand, an Interprocedural Control Flow Graph (ICFG) [17] contains too much information that is not connected to procedure calls.

First, we define the intraprocedural *Component Control Flow Graph* (CCFG) where only call site nodes are considered. Each CCFG represents a procedure and contains one entry node and several component nodes. We get these component nodes, by determining the strongly connected subgraph of the control flow graph of the procedure. The components are connected by control flow edges. We can further reduce this component graph, if we drop out the components with no call sites, and we insert control flow edges among its predecessor and successor components[6]. The remainding nodes are ordered into topological sequences[7]. In the *Interprocedural Component Control Flow Graph* (ICCFG), each procedure is represented by a simple CCFG, and these CCFGs are connected by call edges. There is a call edge from a given component node C to a procedure entry of m if and only if at least one call site of C calls m.

To understand how the ICCFG is built up, let us see the example in Fig. 1. This program executes a function selected by the user. The input of the function is also determined by the user. If the user's function is not defined by the `init` function in the program, the user has to select another function. If the input of the function is not valid (it means that it is not convertible to double), the input will be the default 0 value. In the `init` function, the program initializes the list of the executable functions. We can further complete the defined list with arbitrary functions expanded on a double value. The `readInputs` function reads the user's inputs, while the `checkInputs` function checks whether the function selected by the user is executable by the program. The program executes a well defined function with the help of a function pointer. The allocated memory is freed by the functions `deleteFunctions` and `deleteName` at the end of the program.

Fig. 2 shows the control flow graph of the `main` function, which is the most complex procedure of the program. The strongly connected components of the control flow graph are rounded by broken lines. The components which contain function calls as well are emphasized. The CCFG graph is determined by these components and the control flows among them. These components are the black nodes of the ICCFG graph in Fig. 3. For the sake of simplicity, the figure contains only the entry nodes of the called procedures, but of course, these procedures also have their own components.

There are four kinds of components.

a.) The component contains only one function call, and the call has only one target.

b.) The component contains only one function call, but the call

[6] If two call sites are mutually reachable from each other by control flow edges then they are represented by the same node.

[7] This ordering is important only for our second algorithm, which is introduced in Fig. 5.

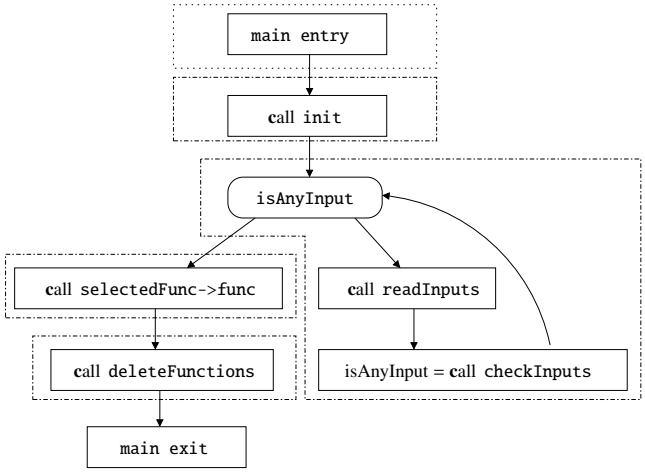

Fig. 2. The CCFG graph of the `main` function of the example program in Fig. 1

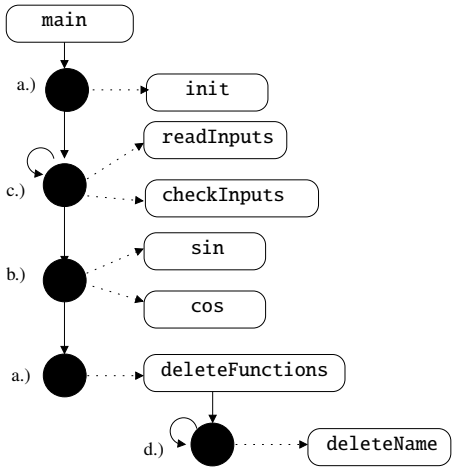

Fig. 3. The ICCFG graph of the example in Fig. 1

has more targets. This situation is due to the fact that the target of the function call is not determined during compilation time. This can be caused by function pointers and virtual function calls. At this point, our graph representation can be very conservative. But the more precise the call graph representation of our program is, the more precise our graph representation is as well. We can improve the precision of our call graph by applying an appropriate pointer analysis algorithm or, in the case of object oriented programs, with the help of Rapid Type Analysis [3].

c.) The component contains one or more function calls which are in the same loop, but these calls have more targets. This situation occurs in a case which is almost the same as point b with the difference that the function call is in a loop. The example in Fig. 3 shows the situation when there are more function calls with different targets. In these cases, all pairs of the targets will be in SEA relation.

d.) The component contains one function call with one target, but the function call is in a loop. In this situation, it is possible that the called function is executed several times, so the called function is in SEA even if the ID relation is not part of the SEA relation.

3.2 Algorithm for computing SEA

In the paper [16], we introduced an algorithm that for a given procedure computes the set of procedures that are in SEB relation with it. In this paper, we give the pair of this algorithm where we determine the set of procedures which are in SEA relation with the given procedure. Due to the features of our graph representation, traversing the edges of our graph representation in the reverse direction is not enough.

The algorithm in Fig. 4 computes the SEA set only for a particular procedure. In the first round, between lines 4 and 7, it collects the procedures that call the given procedure, or rather the components in which the control returns after the execution of the selected procedure. In this way we can collect the RET relations. In the second round, between lines 8 and 14, starting from these components we can collect the procedures which are in SEQ and CALL relations with the given procedure.

Our other algorithm for computing SEA was introduced in [8]. In this algorithm, we compute the SEA sets for all procedures concurrently. However, we can refine our earlier algorithm with the fundamental observation that we can reduce the size of our graph representation not only by the strongly connected components of the control flow graphs, but also by the strongly connected components of the call graph.

Lemma 1 *Let $SCC\_CallGraph$ be a strongly connected component graph of the call graph. Let F and G be two components of the $SCC\_CallGraph$ where procedures $f \in F$ and $g \in G$. If $(f, g) \in SEA$, then for all elements of F and G are in SEA relation.*

Proof. The procedures inside a strongly connected component of the call graph are in transitive call relation. For all $f'(\neq f) \in F$ there is a transitive $CALL$ (or RET) relation among f' and f and the same is realized for all $g'(\neq g) \in G$ and g procedures. If f and g are in CALL, RET or SEQ relation, for all $f' \in F$ and $g' \in G$ are in CALL, RET or SEQ relation, respectively. The realization of the CALL and RET relations comes from the transitivity of these relationships, while the existence of the SEQ relation comes from its definition.

Our modified SEA algorithm in Fig. 5 computes the SEA sets of all procedures at the same time. In the first line, we determine the strongly connected components of the call graph. Lines between 2 and 5 are responsible for determining the CALL and RET relations. This computation starts from the last component of the sequence of the topologically sorted components. For every component we determine the set of components that can be called from the selected component, and the set of components from which the flow of control can return to the selected one. Lines between 6 and 14 determine the SEQ relations appearing among the different call sites of a procedure. The topologically sorted list of components of the procedure is traversed from the

```
program computeSEA(P, f)
input:  P : ICCFG of program P
        f : a procedure in P
output: S : set of procedures that are in SEA
            relation with f
begin
1   Empty S
2   Color components of procedures of P transparent
3   Color components of f to grey
4   Traverse P from entry node of f in backward direction
5       If component c is reached from an entry (not from a component)
        then color the successor components of c to grey.
6       If the entry of the procedure e is reached by the traversal then
        insert e to S.
7       During the traversal each edge may be touched at most once.

8   While there are grey components
9       Let c be a grey component
10      Color c to black
11      Color the uncolored successor components of c to grey.
12      For all g procedures called by c
13          Insert g to S.
14          Color the uncolored components of g to grey.
15  Output S
end
```

Fig. 4. Computation of the SEA relation with graph reachability

first to the last. So, when component c of the procedure is under observation, all p predecessor components of c have been prepared. We determine the set of procedures that is available before calling p or during the execution of p. So, when we extend the SEQ relations in line 14, the *prev* sets of component c contain all the procedures which can be executed before c during the execution of the investigated procedure m. In line 15, the operation union has to consider that the CALL, RET and SEQ relations are determined between components on behalf of the procedures. The reflexivity of the SEA relationship is ensured at this point.

3.3 Comparison of the algorithms

It can be an interesting question whether the algorithm in Fig. 4 or the algorithm in Fig. 5 is more useful for a particular case. It seems that if we want to know the impacts of only few procedures then the algorithm in Fig. 4 is more efficient. However, in many cases we are interested in the determination of the impact of almost every procedure. Although there are many factors of the algorithms that determine their complexity, we try to compare them. First, let us see the steps of the algorithms, which mainly determine their costs:

- Algorithm in Fig. 4
  a) Rows 1-3: initialization
  b) Rows 4-7: traversing all the incoming edges of the entry nodes of procedures and the incoming flow edges of components. This step determines the RET relations of the given procedure.
  c) Rows 8-14: traversing all the outgoing call and flow edges of components and procedure entry nodes starting from the components that are the successors of the components collected by the first traversal of the graph. This step determines the CALL and SEQ relations of the given procedure.
- Algorithm in Fig. 5
  a) Rows 1-5: computing the CALL and RET relations.
  b) Rows 6-14: traversing all components in the procedure and making the Cartesian product of the set of procedures that can be executed before the execution of the component but after entering the actual procedure and the set of procedures called direct or indirect by the given component. Actually these steps determine the SEQ realations that come into existence by the procedure m.

The main difference between the algorithms is that the first one computes the impacts of only one procedure, while the second one computes the impacts of all procedures. So, if we want to know the total cost of the earlier algorithm, we have to multiply its cost with the number of procedures for which we want to compute the impact sets.

```
program computeSEA(P)
input:  P  :  ICCFG of program P
output:  SEA  :  the SEA relation for all procedures
begin
1    SCC_CallGraph := componentQueue(callGraph).
2    for cc := last(SCC_CallGraph) to first(SCC_CallGraph)
3       forall cc_next components of succComponentOf[cc]
4          CALL[cc].insert (cc_next ∪ CALL[cc_next]);
       endforall
5       RET = inv(CALL);
     endfor
6    forall m procedures of the program
7       topOrder := componentQueue(m)
8       for c := first(topOrder) to last(topOrder)
9          prev[c] := ⋃_{p∈previous components of c} prev[p]
10         if c is in loop
11            prev[c] := prev[c] ∪ CALL[c]
           endif
12         SEQ := SEQ ∪ (prev[c] × CALL[c])
13         if c is not in loop
14            prev[c] := prev[c] ∪ CALL[c]
           endif
        endfor
     endforall
15   SEA := CALL ∪ RET ∪ SEQ;
end
```

Fig. 5. Computation of SEA for all procedures

Contrary to the algorithm in Fig. 4, the main advantage of the algorithm in Fig. 5 is that it needs to visit all the procedures only once during the computation, while the other has to traverse the procedures several times when we want to compute the impact sets of several procedures. Anyway, this multiple traversal can be cheaper in many cases than the Cartesian product computation of the algorithm in Fig. 5.

The worst case computational complexities of the algorithms are the following: let n be the number of procedures, k and e be the maximum number of component nodes and edges in the procedures respectively, and m be the maximum number of procedure calls in a component. The first method traverses the graph twice to determine the relations of a given method. Both traversals are linear to the graph size, thus this method has a worst case computational complexity of $O(n \cdot e + n \cdot k \cdot m)$, if we want to compute the SEA relations of all n procedures. The second algorithm first determines the transitive calls of all procedures, then it computes an ordering of the components and performs set operations for each component. If appropriate data structures are used, this is done in $O(n \cdot e + n \cdot k \cdot m)$ time. These complexities are roughly the same as the computational complexity of the detailed SDG-based static slicing algorithm [15]. However, there are significant differences between slicing and our two approaches. The main difference is in the building of SDG and ICCFG. The building of both requires an Interprocedural Control Flow Graph (ICFG). The ICCFG can be easily derived from the ICFG by deleting nodes and performing two depth-first graph traversals for finding strongly connected components. On the other hand, the building of the SDG requires the computation of control and data dependencies, which are additional (also complex) algorithms. Finally, the number (k) of nodes of the graph of the given procedure is also larger in the SDG than in the ICCFG. Thus, the overall computation complexity of the ICCFG is significantly better than that of the SDG.

To return to the comparison of our two algorithms, we analyzed the source of the `gcc` compiler. Of course the worst case computational complexities of the algorithms are almost the same, there could be significant differences in practice. In order to get comparable results, we used the same framework and data representation for handling the recognized relations. We chose the `gcc` for this measurement because its graph representation size was enough to emphasize the differences between the execution times of the algorithms. We executed the algorithm in Fig. 4 for every procedure and we also executed the algorithm in Fig. 5 and its original version.

We found that the total execution time of the algorithm in Fig. 4 was quintuple of the algorithm in Fig. 5, and the execution time of this algorithm was half the execution time of the original algorithm introduced in [8].

Although this practical measurement is not enough to decide which algorithm is better in a particular situation, we can say that the more procedures' SEA sets we want to calculate, the more likely we have to apply the algorithm in Fig. 5.

4 Empirical results

In this section, we try to prove with experimental results that the determination of the SEA/SEB relations is safely usable in the procedure level impact analysis and it can replace the more resource demanding slicing[8]. We compare the SEA/SEB sets of the procedures of the investigated programs with the results of procedure level slicing where the result sets connected to the particular procedure contain the procedures which are reachable from any slice starting from any criterion of the given procedure. Additionally, we will show that an impact analyzer tool, which is built only on a call graph or on a dependence graph which contains only partial information about the program, gives more imprecise results for these programs.

4.1 Computation of recall and precision

During our measurements, we compare the obtained results with the results of slicing[9]. For each procedure, we determine the sets of procedures which are contained by any of the forward (backward) slices starting from the procedure and we compute the SEA/SEB sets of each procedure. For each procedure, these sets contain the procedures which are in SEA/SEB relation with the given procedure.

For a given procedure

- the true positive examples (TP): the procedures which are identified by either the slicer or the SEA/SEB relation,
- the false negative examples (FN): the procedures which are identified by the slicer, but not by the SEA/SEB relation,
- the false positive examples (FP): the procedures which are identified by the SEA/SEB relation, but not by the slicer,
- the true negative examples (TN): the procedures which are identified by neither the slicer nor the SEA/SEB relation.

The recall value:

$$\text{recall} = \frac{TP}{TP + FN}$$

The precision value:

$$\text{precision} = \frac{TP}{TP + FP}$$

A program slice starting from any criterion of the program is easy to obtain by the appropriate traversal of the edges of the system dependence graph of the program [15]. The touched edges are the control-, data-, parameter- and the summary edges. The latter connects the formal input and output parameters of the procedures. Theoretically, these edges could only appear between such nodes among which there is a path determined by control flow edges. Thus, the procedure level results of slicing must be the real subset of the results of the SEA/SEB sets computations, and the recall value must be 100% in every case.

This assumption was not fulfilled in the comparison of the results of forward slices determined by the CodeSurfer program slicing tool and the SEA sets, while it was fulfilled in the backward cases[10]. We investigate the reasons for this in Section 5.

So, in our experiments, we compare only the connection between backward slices and SEB relations. We can do this, since the data could show the same result in the forward case, if the duality of the backward and forward slices was also realized in practice.

4.2 Implementation

So that the SEA/SEB sets computed by our algorithm will be comparable with the results of the CodeSurfer program slicing tool [13], the ICCFG graph, which is the basis of our algorithm, is produced by using the CodeSurfer's Application Programming Interface. Of course, our method is independent of the detailed computations of the CodeSurfer, since it is built only on the call graph and the intraprocedural control flow graphs of the program. This information can be extracted from the program by other tools. For this purpose, we used our self developed Columbus framework [11] in our earlier paper [8]. Fig. 6 shows the computational steps for determining the appropriate graph representations to compute slices and SEA/SEB relations.

CodeSurfer permits the construction of different graph representations according to the presets defined by the user. It is used as the common front end which performs source code parsing and produces the common internal representation, which may be slightly different in the cases of the two dependence computation parts.

We used different presets of the frontend for the two methods in order to gain a more optimal performance for each of the approaches (see Table 1). These presets are different in the SEA/SEB computation and in slicing, because the SEA/SEB computation requires less information and uses less expensive computations than slicing.

In order to determine a program slice starting from a particular program point, one has to determine the control- and data

[8] Of course, it would be much interesting to give an exact mathematical way for this comparison, but it is almost impossible, because the semantic and the structure of the analyzed program strongly affect the difference between the results of slicing and of our approaches. There are programs where the results of the slicing and the SEA computation are the same, while in the worst case the SEA result contains all the procedure of the analyzed programs even if the result of the slicing is more less.

[9] The SEA and SEB relations approximate the forward and the backward slice respectively.

[10] This apparent aberration comes from the fact that the duality of backward and forward slices is not fulfilled in practice by applying the CodeSurfer (2.1p1) program slicing tool.

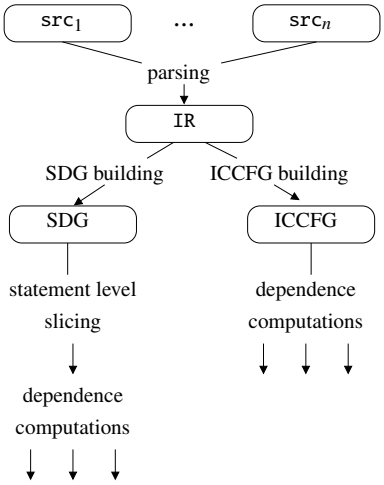

Fig. 6. Experimental tool architecture

Tab. 1. Different presets of the used CodeSurfer

Preset of CodeSurfer	SDG	ICCFG
-control-dependence	yes	no
-data-dependence	yes	no
-compute-gmod	yes	no
-compute-summaries	yes	no
-cfg-edges	no	yes, both directions
-basic-blocks	no	yes

dependencies inside each procedure, the use of global and other variables has to be recorded for every procedure, and the summary edges which connect the formal inputs and outputs of the procedures have to be computed.

For the construction of the ICCFG graph, it is sufficient to give the program call graph and collect the call sites of each procedure with the control flow edges connected them. Thus, the construction of the ICCFG graph is only a simple conversion. We have to compute the topologically ordered sequence of the strongly connected components of the vertices determined by the call sites and we have to note the call edges starting from them, and the flow edges among them.

4.3 Subject programs

For the experiments about precision, we started with the suite of C programs of Binkley and Harman [9], but in some cases, we used different versions. Table 2 lists the subject programs with some related basic data, namely the number of procedures and the lines of code (TL means total lines, while LCode means logical line as provided by the CodeSurfer). In order to find the limits of the different approaches in terms of space and time costs, the efficiency of the methods is verified on several C/C++ large software systems available as open source. The basic features of these systems are listed in Table 3.

4.4 Results

Fig. 7 shows that the call graph only does not give sufficient information about the possible impact sets. The recall values are low in this case, so in many cases, the impact sets determined by

Tab. 2. C language test programs for precision measurements

Program	Number of procedures	TL	LCode
time v1.7	12	1 314	757
replace v?	21	563	512
compress v?	24	1 937	1 335
wdiff v0.5	27	1 862	1 080
which v2.17	28	1 989	1 246
acct v6.3	50	3 510	1 996
termutils v2.0	57	3 685	2 518
barcode v0.98	62	3 885	2 331
indent v2.29	111	11 539	7 582
ed v0.8	120	3 052	2 267
EPWIC v?	149	9 597	5 249
flex v2.4.7	152	14 184	9 134
byacc v1.9	178	3 553	2 737
diffutils v2.8	192	15 022	9 735
bc v1.06	204	7 794	5 290
userv v0.95.0	239	7 909	6 016
copia v?	242	1 168	1 085
gnuchess v5.07	261	16 533	11 045
tile-forth v2.1	286	5 730	3 549
li v?	357	7 597	4 793
espresso v?	361	22 050	21 780
go v?	372	29 629	22 118
ijpeg v?	467	28 185	15 253
ctags v5.0	518	13 750	10 018
sendmail v8.14	548	123 965	77 950
findutils v4.2.31	608	41 661	27 261
a2ps v4.13	902	54 954	33 573
gnubg v1.2	192	6 705	4 330
gnugo v3.6	2 188	151 376	110 631

Tab. 3. C/C++ language test systems for efficiency measurements

System	Number of procedures	TL	LCode
valgrind 3.3.0 (C)	5 318	228 763	141 631
gdb 6.7.1 (C)	8 095	473 793	303 552
gcc 4.0 (C)	16 108	1 052 353	725 620
mozilla 1.6 (C++)	83 432	2 382 459	1 414 946

the call graph only can hardly approximate the real impact sets. Fig. 8 compares the average size of the call, control, slice and the SEA/SEB relations connected to the individual procedures. The control dependence is a little bit more precise than the call relation, because in many cases the execution of a procedure can determine the execution of another procedure, even if there is no call relation between them. However, Fig. 8 shows that the control dependence information is not enough either to predict the real impact set of a particular procedure. This is a very important observation, because it means that significant parts of the impacts are caused only by data dependencies.

According to our expectations, the recall values of the impact sets determined by the SEB relations are 100% in all cases, and the precision values are also high. Apart from the case when the precision is only about 67%, the precision values are between

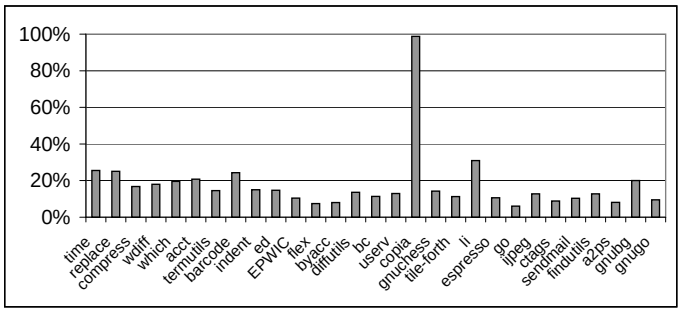

Fig. 7. The recall values of impact sets determined by only the call graph.

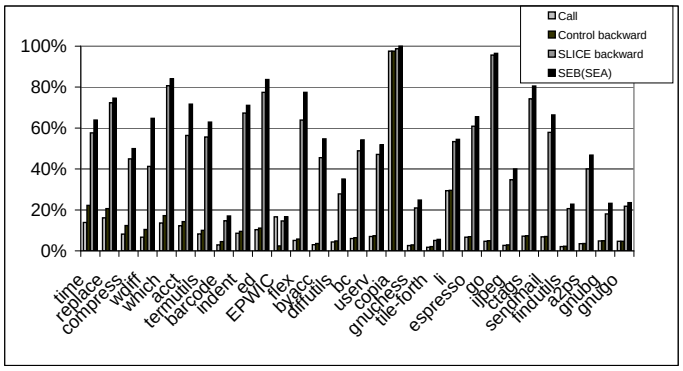

Fig. 8. Comparison of the size of the average call, control, slice and the SEA/SEB relations of procedures

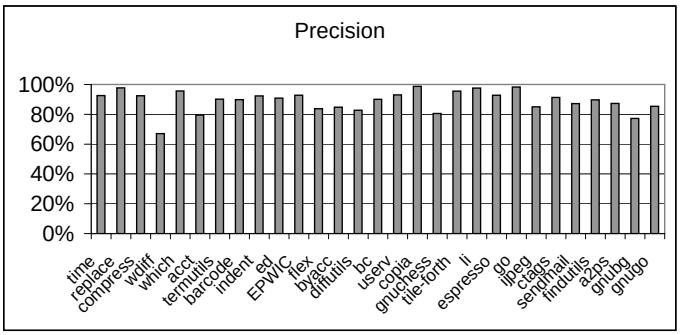

Fig. 9. The precision of the SEB sets relative to the backward slices.

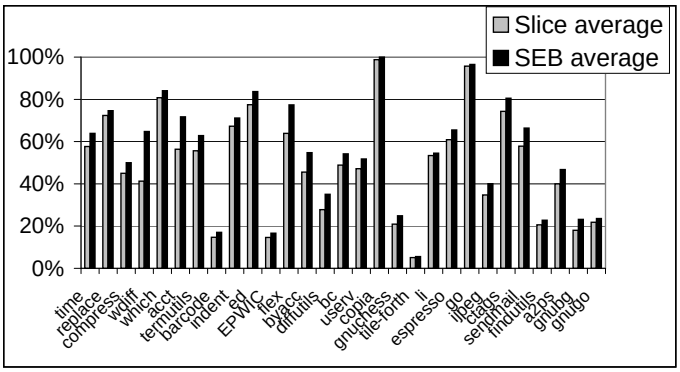

Fig. 10. The data of Table 4

77,28% and 98,77% as it is readable from Fig. 9. We can say that the SEB relations nicely approximate the backward slice results. The main disadvantage of the static slice is said to be the fact that it contains almost the whole program, so the use of the static slice is not so effective. In Table 4, by investigating all procedures of certain programs, we collect the average percentage of the procedures which are in any of the slices starting from any criterion of the given procedure. In this table, we also introduce the average size of the SEB relations of procedures in a program. It is easy to see that there are some cases where the average slice size becomes almost the same as the program size, but this is not typical.

Up to now, the examined data have shown the average sizes of slices and the SEB sets connected to the certain procedures. However, the SEB sets approximate the slices well, not only on average, but for every procedure as well. Fig. 11 shows the histogram of the differences between backward slice sizes and the sizes of the SEB sets of each procedure of each program. As it is readable from the figure, these differences are very small in most cases. Although there are cases where the difference reaches 48% of the size of the given program, the number of these situations is irrelevant according to the number of all procedures. Moreover, having investigated the situations where the difference was greater than 25%, we found that only three programs – `wdiff, barcode, gnuchess` – were responsible for these cases.

The fact that the determination of the SEB relation approximates accurately the slice sizes is not enough in itself. The main advantages of our method and our graph representation are that they are much simpler than the graph representation of the program, which is suitable on slicing. Both slicing and determining the SEB are reachability problems on an appropriate graph representation. Due to the much bigger representation, the graph traversal can be much more time consuming in the case of slicing. Although so far, slicing has been somewhat more expensive than the computation of SEB sets in the case of most programs, using a slicer can be convenient even with the computers used nowadays. However, in the case of programs with thousands of procedures and millions of lines of code where slicing is not feasible, we can use our technique efficiently.

The two program representations used in our case study share the same structure on the highest level, namely both of them include a node for each procedure of the program. However, there are significant differences in these representations regarding the amount of data to be stored for a procedure. Table 5 contains the relevant numbers. It can be easily deduced that ICCFGs require a significantly smaller amount of nodes and edges and the difference is about two degrees of magnitude. The CodeSurfer could not build up the system dependence graph of the biggest program, so the corresponding data are missing from the table.

Table 6 investigates major real applications. It compares the building times of the system dependence graph and the ICCFG graph, which are the basis of slicing and the SEB computations respectively. We made our experiments on an AMD Opteron 2.2 GHZ processor with 4G memory.

Tab. 4. The average size of backward slices and the average size of the SEB sets.

Programs	Backward slice sizes	SEB sizes
a2ps	39,99%	46,78%
acct	56,40%	71,72%
barcode	14,72%	17,07%
bc	48,88%	54,22%
byacc	45,55%	54,74%
compress	44,97%	50,00%
copia	98,76%	99,99%
ctags	74,29%	80,55%
diffutils	27,80%	35,09%
ed	77,42%	83,70%
EPWIC	14,65%	16,63%
espresso	60,88%	65,51%
findutils	20,63%	22,77%
flex	63,92%	77,41%
gnubg	18,03%	23,20%
gnuchess	20,96%	24,84%
gnugo	21,84%	23,56%
go	95,69%	96,49%
ijpeg	34,71%	40,05%
indent	67,32%	71,13%
li	53,38%	54,50%
replace	72,34%	74,60%
sendmail	57,86%	66,39%
termutils	55,68%	62,88%
tile-forth	5,12%	5,61%
time	57,64%	63,89%
userv	47,14%	51,81%
wdiff	41,29%	64,75%
which	80,74%	84,06%

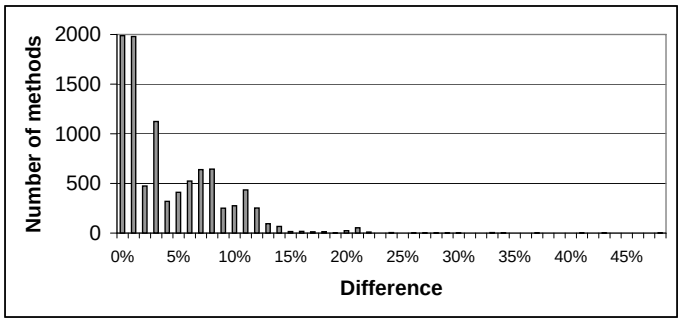

Fig. 11. Differences of the SEB sets and the backward slices of the procedures of all programs.

Tab. 5. The sizes of the different graph representations

valgrind	SDG	vertices	1 920 150
		edges	6 947 024
	ICCFG	vertices	154 509
		edges	179 626
gdb	SDG	vertices	10 086 409
		edges	48 876 108
	ICCFG	vertices	160 340
		edges	185 611
gcc	SDG	vertices	18 775 143
		edges	81 492 908
	ICCFG	vertices	467 185
		edges	584 972
mozilla	SDG	vertices	N/A
		edges	N/A
	ICCFG	vertices	1 587 499
		edges	1 723 611

5 Duality of the backward and forward slices

Even though from the point of view of impact analysis the comparison of the forward slices and the SEA relation would be a more obvious choice in the empirical measurements, we took the reverse direction. The main reason for this choice was that the imprecision of the forward slice caused some differences which distorted our results. Actually, a well functioning forward slicer is the dual pair of the backward slicer. It means that if the forward slice of the program point A contains the program point B, then the backward slice of the program point B contains the program point A. Our measurements investigate the slices of all program points as the criteria of the slices, so if duality was realized, the average size of the result sets would be the same in the two directions. In this section, we investigate how it is possible that this duality feature is not fulfilled.

The slicing algorithm of the CodeSurfer is based on the algorithm of Ball and Horwitz [5]. For generating executable slices, it augments the original control flow graph of the program with additional control flow edges which, in the case of jump or abort statements (`break, continue, goto, return` ...), point to those statements where the control would follow the execution if there was not a jump statement in that point. So, the latter statements which the control does not get into are in essence control dependent on the jump or abort statements. This method is suitable for slice computation where the slice has to be executable and semantically the same as the original program.

Bent *et al.* compared one of the earlier versions of the CodeSurfer with their own slicer called Sprite [6]. They investigated how a more precise dependence determination could affect the size of program slices. In most cases, the CodeSurfer was much more precise. However, according to the fact that the CodeSurfer makes executable slices, in some cases, the slices also contain statements which cannot affect the behaviour of the given program point and could be dropped out of the slice.

The backward slice from the 9th line of the example in Fig. 12 originally contains – in the investigation of Bent *et al.* – the 5th and 6th lines. Of course, these lines do not have any effect on the criterion, and without these lines, the slice is also executable and semantically has the same behavior as the original program in the statements of line 9. Similarly the forward slice from the condition of line 5 contains the statements of line 9.

This problem is originated in the algorithm of Ball and Horwitz [5]. Consider the control flow graph of the procedure `foo` and complete it with the control flow edges given by the algorithm. Build the program dependence graph by taking into consideration the augmented edges by which *false dependent con-*

Tab. 6. Building times of graph representations

System	Parsing time	SDG building time	ICCFG building time
valgrind	5 min	16 min	2 min
gdb	8 min	124 min	4 min
gcc	69 min	571 min	11 min
mozilla	113 min	N/A	54 min

```
1    int foo(int i){
2      int result := 0;
3      switch(i) {
4        case 1:
5          if (hoo())
6            return 0;
7          break;
8        case 2:
9           result = goo();
10          break;
11     }
12     return result;
13   }
```

Fig. 12. Example according to Bent *et al.*

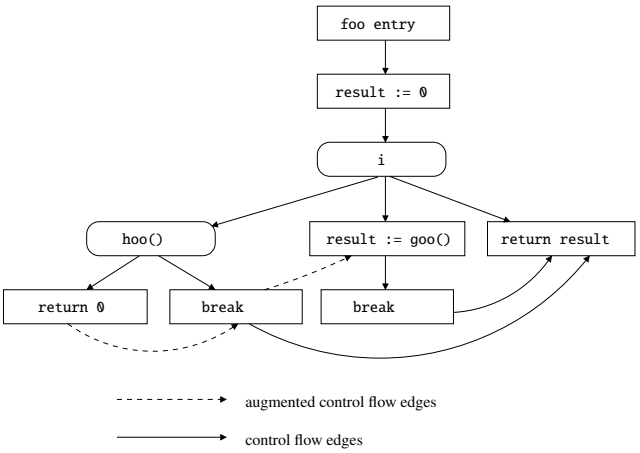

Fig. 13. The control flow graph of the code in Fig. 12

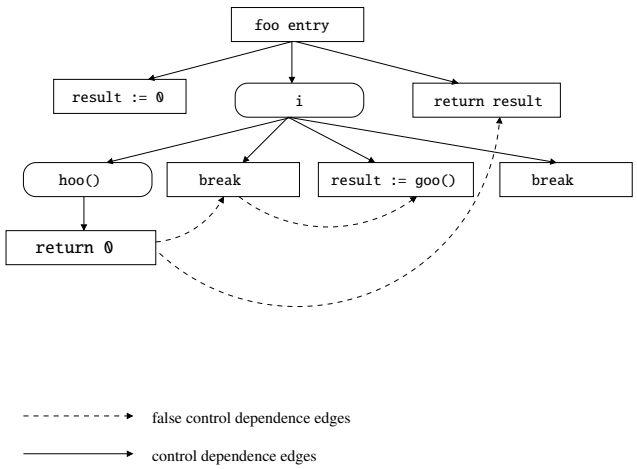

Fig. 14. The program dependence graph of the code in Fig. 12

trol flow edges can be imported. Fig. 13 and Fig. 14 show these graphs of the example in Fig. 12, respectively [11].

Tab. 7. Test programs where the forward slices contain redundant elements.

Program	Differences between forward and backward slices	Erroneously classified elements in the forward slices
termutils	0,15%	0,276%
ed	0,39%	0,482%
diffutils	0,50%	1,76%
espresso	5,35%	8,027%
sendmail	0,02%	0,028%
a2ps	4,25%	9,423%
gnubg	0,43%	2,24%
gnugo	0,69%	2,764%

This false control dependence determined by Ball *et al.* is a non transitive relation, while the determination of a slice remains a simple reachability problem over the augmented program dependence graph as it was in the algorithm of Horwitz *et al.* [15]. Since false control dependence is not a transitive relation, the slice should not contain a program point if there is a path between the criterion and the program point where the false control dependence edges follow each other. The elimination of these situations could reduce the size of the slice and the slice would still remain executable.

Although the present version of the CodeSurfer (2.1p1) pays attention to this situation in the case of backward slices, it does not in the case of forward slices. According to the faith of the GrammaTech Inc., it will be corrected in the next release of the CodeSurfer.

We have to note that Harman and Danicic published an algorithm for slicing unstructured programs [14] and they proved that their solution is more precise than the algorithm of Agrawal [1] and the algorithm of Ball and Horwitz. Nevertheless, they only dealt with backward slices as well and they did not investigate whether the adaptation of their algorithm in the forward direction gave the dual pair of the backward slice or not.

5.1 The effect of this problem on the experimental results

The effect of the above introduced problem is that the duality among the backward and forward slices is not fulfilled in some cases. So, it is possible that there is a program point A whose forward slice contains the program point B, while the backward slice of the program point B does not contain the program point A. In addition, this problem decreases in the comparison of forward slices and SEA sets. In some cases, there are procedures whose slices contain such a procedure that is not in the SEA set of the given procedures[12].

Table 7 contains the programs of our tests where this problem appeared. Since we made slices from all potential vertices of the program, our original assumption was that the summary of

[11] Data dependencies are not represented. These are no effect to this problem.

[12] This is a rightful trouble. The SEA set of a procedure does not contain rightfully the procedures which are not reachable from the given procedure by control flow edges.

the size of forward slices and the summary of the size of backward slices would be the same. The first column of the table shows for each procedure with what percentage the forward slice is larger than the backward slice in general. The second column describes for each program what percentage of the forward slices is found false in the slice by SEA.

Altogether there were only 8 programs where the problem appeared. In many cases, the number of the incorrectly identified elements in the slices was very low, but in some cases, the number of unnecessary elements in the slices is remarkable.

6 Conclusion

In this paper, we present two methods to compute impact sets for procedures with the help of our ICCFG graph representation of the program. The ICCFG graph is based on the call graph of the program and on the reduced control call graphs of each procedure. We have proved with experimental results that the precision of our methods is slightly more inaccurate than slicing and it is efficiently applicable on large programs, too.

The investigation of the experimental results also reveals the fact that the more expensive computations do not have the appropriate benefits on any level of abstraction. Although slicing can be more precise on statement level, on a higher level of abstraction e.g. on procedure level, a not so precise but more effective method such as the SEA computation can be more useful as an implementation of impact analysis. Additionally, in a complex computation like slicing, it is easier to make a mistake just as the slicer did in our case when handling arbitrary control flows. Moreover these inaccuracies of a complex algorithm can be completed with any other intentional imprecision for the sake of efficiency. For example, the handling of points-to information, arrays, structure fields, etc. can be different during slicing depending on whether the precision or the efficiency is more important. In these cases, it is increasingly more likely that the originally more imprecise but more effective algorithm has the same results with much less effort.

References

1 **Agrawal H**, *On slicing programs with jump statements*, PLDI '94: Proceedings of the ACM Sigplan 1994 Conference on Programming Language Design and Implementation, 1994, pp. 302-312, DOI 10.1145/773473.178456, (to appear in print).

2 **Apiwattanapong T, Orso A, Harrold M J**, *Efficient and precise dynamic impact analysis using execute-after sequences*, Proceedings of the 27th International Conference on Software Engineering (ICSE'05), May 2005, pp. 432-441, DOI 10.1145/1062455.1062534, (to appear in print).

3 **Bacon D F**, *Fast and effective optimization of statically typed object-oriented languages*, Ph.D. Thesis, 1997. Chair-Susan L. Graham.

4 **Badri L, Badri M, St-Yves D**, *Supporting predictive change impact analysis: A control call graph based technique*, Apsec '05: Proceedings of the 12th Asia-Pacific Software Engineering Conference (APSEC'05), 2005, pp. 167-175, DOI 10.1109/APSEC.2005.100, (to appear in print).

5 **Ball T, Horwitz Susan**, *Slicing programs with arbitrary control-flow*, Automated and algorithmic debugging, 1993, pp. 206-222, DOI 10.1007/BFb0019410, (to appear in print), `citeseer.ist.psu.edu/article/ball92slicing.html`.

6 **Bent L, Atkinson D, Griswold W**, *A qualitative study of two whole-program slicers for C*, `citeseer.ist.psu.edu/bent01qualitative.html`.

7 **Beszédes Á, Gergely T, Faragó Sz, Gyimóthy T, Fischer F**, *The dynamic function coupling metric and its use in software evolution*, Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR 2007), 2007mar 21, pp. 103-112, DOI 10.1109/CSMR.2007.47, (to appear in print).

8 **Beszédes Á, Gergely T, Jász J, Tóth G, Gyimóthy T, Rajlich V**, *Computation of static execute after relation with applications to software maintenance*, Proceedings of the 2007 IEEE International Conference on Software Maintenance (ICSM'07), October 2007, pp. 295-304, DOI 10.1109/ICSM.2007.4362642, (to appear in print).

9 **Binkley D, Harman M**, *A large-scale empirical study of forward and backward static slice size and context sensitivity*, Proceedings of the International Conference on Software Maintenance (ICSM'03), September 2003, pp. 44-53, DOI 10.1109/ICSM.2003.1235405, (to appear in print).

10 **Bohner Sh A, Arnold R S. (eds.)**, *Software change impact analysis*, IEEE Computer Society Press, 1996.

11 **Ferenc R, Beszédes Á, Tarkiainen M, Gyimóthy T**, *Columbus – reverse engineering tool and schema for C++*, Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2002), October 2002, pp. 172-181.

12 **Gallagher K B, Lyle J R.**, *Using program slicing in software maintenance*, IEEE Transactions on Software Engineering **17** (1991), no. 8, 751-761, DOI 10.1109/32.83912.

13 *GrammaTech's CodeSurfer*, GrammaTech, Inc., `http://www.grammatech.com/products/codesurfer`.

14 **Harman M, Danicic S**, *A new algorithm for slicing unstructured programs*, Journal of Software Maintenance **10** (1998), no. 6, 415–441, DOI 10.1002/(SICI)1096-908X(199811/12)10:6<415::AID-SMR180>3.0.CO;2-Z.

15 **Horwitz S, Reps T, Binkley D**, *Interprocedural slicing using dependence graphs*, ACM Transactions on Programming Languages and Systems **12** (1990), no. 1, 26-61, DOI 10.1145/77606.77608.

16 **Jász J, Beszédes Á, Gyimóthy T, Rajlich V**, *Static Execute After/Before as a replacement of traditional software dependencies*, Proceedings of the 2008 IEEE International Conference on Software Maintenance (ICSM'08), September 2008, DOI 10.1109/ICSM.2008.4658062, (to appear in print).

17 **Landi W, Ryder B G.**, *Pointer-induced aliasing: a problem taxonomy*, Popl '91: Proceedings of the 18th ACM Sigplan-Sigact Symposium on Principles of Programming Languages, January 1991, pp. 93-103, DOI 10.1145/99583.99599, (to appear in print).

18 **Law J, Rothermel G**, *Incremental dynamic impact analysis for evolving software systems*, Proceedings of the 14th International Symposium on Software Reliability Engineering, November 2003, pp. 430-441, DOI 10.1109/ISSRE.2003.1251064, (to appear in print).

19 ———, *Whole program path-based dynamic impact analysis*, Proceedings of the 25th International Conference on Software Engineering (ICSE'03), May 2003, pp. 308-318, DOI 10.1109/ICSE.2003.1201210, (to appear in print).

20 **Orso Al, Apiwattanapong T, Harrold M J**, *Leveraging field data for impact analysis and regression testing*, Proceedings of the 11th ACM Sigsoft Symposium on Foundations of Software Engineering held jointly with 9th European Software Engineering Conference (ESEC/FSE'03), September 2003, pp. 128-137.

21 **Ren X, Shah F, Tip F, Ryder B, Chesley O**, *Chianti: A tool for change impact analysis of Java programs*, Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'04), October 2004, pp. 432-448.

22 **Ryder B G**, *Constructing the Call Graph of a Program*, IEEE Transac-

tions on Software Engineering **SE-5** (May 1979), no. 3, 216-226, DOI 10.1109/TSE.1979.234183.

23 **Szegedi A, Gyimóthy T**, *Dynamic slicing of Java bytecode programs*, Proceedings of the Fifth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'05), September 2005, pp. 35-44, DOI 10.1109/SCAM.2005.8, (to appear in print).